WinDbg & LLDB cheatsheet for .NET

Starting

	WinDbg	LLDB
Start	`windbg {executable} [{args}]` or File menu	`lldb {executable} [-- {args}]` or `(lldb) file {executable}`
Attach	`windbg -p {pid}` or File menu	`lldb --attach-pid {pid}` or `(lldb) process attach -p {pid}`

Symbols and modules

	WinDbg	LLDB
(Re)load symbols	`ld {module-name}`	`target symbols add {symbol-file-path}`
List modules	`lm`	`image list`
Dump module information	`lmvm {module-name}`	`image dump [symtab|sections|ast|symfile|line-table] {module-name}`
Resolve native function address	`x {module-name}!{function}` (wildcards accepted)	`image lookup -vn {function}` (`-r` for Regex search)
Find nearest symbol	`ln {address}`	`image lookup -va {address}`

Processes and threads

	WinDbg	LLDB
Show processes	`|`	debugs single process
Switch to process	`|{process-num}s`	
List threads	`~`	`thread list`
Select thread	`~{thread-num}s`	`thread select {thread-num}`
Execute on all threads	`~*{command}` or `~*e!{ext-command}`	
Enable child process debugging	`.childdbg 1`	

Program execution

	WinDbg	LLDB
Dissasemble	`u {address}`	`disassemble [-s {address}]` (alias: `di`)
Dissasemble function	`uf {address}`	`disassemble -n {function-name}`
Continue	`g`	`{thread|process} continue`
Step out	`gu`	`thread step-out`
Step into	`t`	`thread step-in (s)` `thread step-inst (si)`
Step over	`p`	`thread step-over`
Continue until address	`{g|t|p}a {address}`	`thread until -a {address}`

Breakpoints

	WinDbg	LLDB
List breakpoints	`bl`	`break[point] list`
Create breakpoint	`bp {address|function-name} [{command}]`	`break[point] set -a {address}` `break[point] set -n {function-name}`
Enable/ disable breakpoint	`b{e|d} {breakpoint-number}`	`break[point] {enable|disable} {breakpoint-number}`
Delete breakpoint	`bc {breakpoint-number|*}`	`break[point] delete {breakpoint-number}`
Create data breakpoint	`[~{thread-num}]ba {access}{size} {address} [{command}]` (fe. `ba r4 0x12345`)	`watch[point] {address} [-w {access}] [-s {size}]`

Call stack

	WinDbg	LLDB
Native call stack	`k [{number-of-frames}]`	`thread backtrace [-c {number-of-frames}]` (alias: `bt`)
All native call stacks	`~*k`	`thread backtrace all`
Switch to stack frame	`.frame [/c] {frame-number}`	`frame select {frame-number}`

Evaluating

	WinDbg	LLDB
Display local variable	`dv [{pattern}]`	`frame variable [{variable-name}]`
Evaluate expression	`? {expression}` and `?? {c++-expression}`	`print {expression}`
Display register	`r [{registry-name}]`	`register read [-all|rax|rsp|....]`

Memory operations

	WinDbg	LLDB
Dump memory	`d{format} {address}` (`.hh d` to list formats)	`memory read [-f {format}] {address}` (`memory read -f ?` to list formats)
Dump memory (with layout)	`dt {type} {address}`	`memory read -t {type} {address}`
Details about memory region	`!address {address}`	`memory region {address}`
Edit memory	`e{format} {address} {value(s)}`	`memory write [-f {format}] {address} {value}`
Display addresses and symbols referencing them, if any	`!address {address}`	

Debugging events

WinDbg

- `sxe [-c {cmd1}] [-c2 {cmd2}] {exception|event}` - enable a given exception or event (1st chance), for example, sxe clr to enable 1st chance notification for managed exceptions
- `sxd {exception|event}` - enable only 2nd chance notification (more at .hh sx)
- gn to mark an exception as handled

LLDB

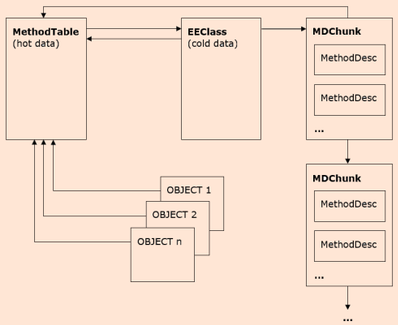
- `process handle [-n {should-notify}] [-s {should-stop}] [-p {should-pass-to-debuggee}] {signal}` - configure unix signal handling
- `breakpoint set -E C++` - break on all C++ exceptions ('1st chance')

.NET SOS extension

	WinDbg	LLDB
Install	It's already in the CLR folder	`dotnet tool install -g dotnet-sos` `dotnet-sos install`
Load	`.loadby sos coreclr` (.NET Core) `.loadby sos clr` (.NET 4.x)	Installer adds loading automatically
Help	`!sos.help` `!sos.help {command}`	`soshelp` `soshelp {command}`
Enable/ disable breakpoint	`b{e|d} {breakpoint-number}`	`break[point] {enable|disable} {breakpoint-number}`
Delete breakpoint	`bc {breakpoint-number|*}`	`break[point] delete {breakpoint-number}`

.NET SOS extension

Metadata



- `dumpdomain` - dump information about AppDomains
- `dumpmt {address}` - dump a Method Table
- `dumpclass {address}`
- `dumpmd {address}`
- `name2ee {module}!{type-or-method}` - resolve a class name into MT or a method name into MD

Managed code breakpoints

- `bpmd {module} {method}` or `bpmd -md {md}` - create a method breakpoint
- `bpmd {source-file}:{line-number}` - create a breakpoint in the source code file
- `bpmd -list` - list pending breakpoints
- `bpmd -clear {breakpoint-number}` - remove a pending breakpoint
- `bpmd -clearall` - remove all pending breakpoints

Managed heap

- `eeheap [-gc] [-loader]` - show information about the internal CLR memory
- `dumpheap -stat` - show managed heap statistics
- `dumpheap -mt {mt}` or `dumpheap -type {typename}` - dump objects of a specific type
- `dumpobj {address}` - dump a managed object
- `gcroot {address}` - look for references (or roots) to an object
- `finalizequeue` - show all objects registered for finalization

Threads and calls stacks

- `!threads` (WinDbg) / `clrthreads` (LLDB) - list all the managed threads
- `clrstack` - show the managed call stack of the current thread
- `dumpstack` - show the complete call stack (native + managed)
- `dso` - dump managed objects referenced in the call stack
- `pe` - show exception details

IL and assembly

- `dumpil {md}` - dump the IL code of the specified method
- `ip2md {address}` - match the assembly instruction address with MD
- `u {md|address}` - disassembly a method or a code address (with managed metadata annotations)

DiagnosticsExpert.com

@konradkokosa & @lowleveldesign